<알고리즘 Mix Tape 03>

6장 Hash 테이블 (170508 to 170510)

우리가 고속버스를 타든, 비행기를 타든 간에 좌석에 번호가 있기 마련이다. 고속버스나 비행기 예약으로 예를 들 수 있다면 우리가 앉는 자리는 사전 예약하는 사람이 좌석을 지정할 수 있는 방법이 있지만, 예약 시스템을 통해서 임의로 떨구는 경우가 있다. 이러한 원리를 접목시킨 자료구조가 바로 Hash 테이블로 살펴볼 수 있다. 이번 요약정리에서는 인덱스를 정해줌으로서 테이블을 채워 나가는 원리인 Hash 테이블에 대해서 공부를 해보는 기회를 가져보도록 하겠다.

1. Hash Table(해시 테이블)

우선 우리가 여태껏 공부했던 자료구조들을 통해서 저장과 검색의 시간 복잡도에 대해서 복습을 해보고 넘어가자.

-> 순차 리스트(Array List), 연결 리스트(Linked List)
O(n)
-> 이진 탐색 트리(Binary Search Tree)
O(log n). 하지만 편향 이진 트리이면 O(n)...
-> 자가 균형 이진 탐색 트리(Self-Balanced Binary Search Tree)
O(log n)
-> B-Tree
O(log n). 그렇지만 평소에 이진 탐색 트리보다 시간이 빨라지는 건 사실이다.

그렇지만 해시 테이블을 공부해보게 된다면 시간 복잡도가 O(1)로 바뀌는 새로운 경험을 하게 될 것이다.

1-1) Hash Table은 무엇인가?

우리가 데이터를 배열에 저장하였을 경우에는 인덱스를 중점으로, 연결 리스트에 저장하였을 경우에는 노드를 중점으로 저장을 하였다. 그렇지만 해시 테이블은 데이터와 인덱스를 역지사지로 생각을 하는 개념으로 접근 해야겠다. 원소의 값을 통해서 저장되는 위치에 대해 계산을 해서 O(1)의 시간 복잡도로 삽입, 삭제, 검색 기능을 이용할 수 있을뿐더러 자료 탐색을 신속하게 해준다. 하지만 해시 테이블은 애석하게도 공간 복잡도는 O(n)을 잡아먹기 때문에 필요한 데이터들에 대해서만 신속하게 검색을 하는 경우에 매우 유용하게 쓴다고 생각하면 되겠다.

1-2) 해시 테이블의 간단한 사례

아래의 코드를 통해서 해시 테이블이 어떻게 작동을 하는지에 대해 간략하게 짚고 넘어가는 기회를 갖자. 이 코드는 예를 들어 0부터 50까지의 int 형 정수에 대한 데이터를 빠르게 불러오기 위해서 쓰이는 해시 테이블을 만들어보면 아래와 같이 만들 수 있겠다.

```
public class Hash_01{
  static class HashTable{
    int[] a;
  public HashTable(int n){ a=new int[n+1]; } // 0
  public void add(int value){ a[value]=value; } // 1
  public void remove(int value){ a[value]=0; } // 2
  public boolean contains(int value){ // 3
      // 오른쪽 위에서 계속됨...
```

```
      return a[value]==value;
    }
  }
  public static void main(String[] args){
    int maxValue=50;
    HashTable hash=new HashTable(maxValue);
    int[] data={10, 15, 20, 30, 35, 40, 45};
    for(int k=0;k<data.length;k++){
      hash.add(data[k]);
    }
    for(int k=1;k<maxValue;k++){
      if(hash.contains(k)){
        System.out.println(k);
      }
    }
  }
}
```

0 : 우선 해시 테이블을 새로 만들기 위해서 0부터 50까지 데이터를 저장하기 위해서는 어쩔 수 없이 크기를 51로 초기화를 해줘야 된다. int 형 배열을 51개나 가지고 있으니 해시 테이블은 이미 주머니가 빵빵한 셈이라고 보면 되겠다.
1 : 여기서 add라는 메소드는 해시 테이블에 데이터를 추가하는 과정이다. 비록 같은 값들에 대해 같은 인덱스로 저장을 하는 모습처럼 보여서 쉽게 생각할 수도 있는데 이는 충돌에 대해서 생각을 하지 않았을 뿐더러 공간에 대해서도 고려를 안 했기 때문에 이처럼 저장하는 일이 없도록 해야겠다. 이에 대해서는 해시 테이블의 충돌을 막는 과정에서 살펴보도록 하자.
2 : remove 메소드는 말 그대로 데이터를 삭제하는 과정이라고 생각하면 되겠다. 어자피 같은 값에 같은 인덱스를 0으로 바꾸는 과정이다.
3 : contains라는 메소드를 통해서 실질적으로 해시 테이블에 가진 원소들의 요소들에 대해서 부울 변수로 통해 얻어올 수 있다. 이는 맨 마지막에 for 문을 참조하면 되겠다.

그렇지만 실제로 이렇게 초딩처럼 해시 테이블을 쓰는 사람들은 거의 없을 것이다. 또한 51이상의 int형 숫자들이 들어오게 된다면 ArrayBoundsException이 걸릴 게 뻔하다. 이에 대해서 해결하기 위하여 % 연산자를 적극 활용해보도록 하자. 이는 여러분들이 생각했듯이 나머지(MOD) 연산자, 맞다.

아래와 같이 저장된 데이터들에 대해서 테이블의 크기를 소수(Prime Number) 중의 하나인 11로 가정을 하고 저장을 시켜보겠다.
10 -> 10 mod 11 = 10번 인덱스
15 -> 15 mod 11 = 4번 인덱스
20 -> 20 mod 11 = 9번 인덱스
25 -> 25 mod 11 = 3번 인덱스
30 -> 30 mod 11 = 8번 인덱스
40 -> 40 mod 11 = 7번 인덱스
44 -> 44 mod 11 = 0번 인덱스
이처럼 저장하기 위해서는 방금 전에 살펴본 메소드에 대해서 수정을 할 필요가 있다. 다음 쪽 문장을 살펴보도록 하자.
-> Next Page ->

```
public class Hash_02{
  static class HashTable{
    int[] a;
  public HashTable(){ a=new int[11]; } // 0
  public void add(int value){
    a[value%11]=value; // 1
  }
  public void remove(int value){
    a[value%11]=0; // 2
  }
  public boolean contains(int value){
    return a[value%11]==value; // 3
  }
 }
 public static void main(String[] args){
   HashTable hash=new HashTable();
   int maxValue=50;
   int[] data={10, 15, 20, 25, 30, 40, 44};
   for(int k=0;k<data.length;k++){
     hash.add(data[k]);
   }
   for(int k=0;k<=maxValue;k++){
     if(hash.contains(k)) System.out.println(k);
   }
 }
}
```

0 : 아까와 달리 배열의 크기를 입력받지 않고 임의의 소수(Prime Number)에 대해서 초기화를 해준 모습을 볼 수 있겠다. 근데 여기서 왜 소수를 써야 효율적으로 돌아갈까? 이에 대해서는 뒤에서 설명하겠다.
1 : add 메소드에서 % 연산자를 통해서 추가하려는 데이터에 대해서 나머지를 인덱스로 활용해서 저장을 하는 모습을 볼 수 있겠다.
2 : 또한 삭제 메소드에 대해서도 %를 통해서 데이터를 삭제하는 모습을 보여주는데 여기서 아이러니한 점이 있을 것이다. 방금 추가된 데이터들에 대해서 참고를 한다면 20과 같은 나머지로 떨어지는 숫자인 31를 넣으면 20이 삭제된다는 점이다. 이를 해결하기 위해서는 충돌을 해결하는 부분을 참고하자.
3 : 물론 contains는 문제없이 정상적으로 돌아간다. 여기서 나머지도 같아야 되고 추가된 값들도 살펴보면 되지만 충돌을 고려해야 한다면 올바른 메소드는 아니다.
아까와 같은 데이터들을 간이 해시 테이블에 저장하면 아래와 같이 저장이 되겠다.

0	1	2	3	4	5	6	7	8	9	10
44			25	15			40	30	20	10

이처럼 나머지 연산자를 이용하면 효율적으로 저장을 할 수 있는데 아까 언급했듯이 나머지가 같은 데이터(9, 20, 31, 42 등등)에 대해서 어떻게 해결을 해야 할까? 이는 충돌이란 개념을 이용해서 해결을 해야겠다.

1-3) 충돌을 해결합시다
나머지가 중복되는 현상을 충돌이라고 부른다. 충돌을 해결하는 방안에 대해서는 일단 원래 저장해야 하는 인덱스의 다음 인덱스 자리를 참고하는 원리로 생각을 염두하고 넘어가보자. 아까 고속버스 표를 사는 경우를 생각해볼 수 있는데 창가에 앉아서 에어컨을 쐬면서 가고 싶은데 이미 예약한 사람이 있으면 옆자리나 다른 자리로 배정되는 현상으로 보면 쉽다. 실질적으로 다음 자리에 저장하는 방법에 대해서는 여러 가지가 있다.
아까와 같이 저장된 데이터들에 대해서 테이블의 크기는 그대로 11, 다만 나머지를 중복해서 저장을 하는 경우를 살펴보도록 하자.
10 -> 10 mod 11 = 10번 인덱스
15 -> 15 mod 11 = 4번 인덱스
21 -> 21 mod 11 = 10번 인덱스
26 -> 26 mod 11 = 4번 인덱스
31 -> 31 mod 11 = 9번 인덱스
37 -> 37 mod 11 = 4번 인덱스
우선 10과 15는 각각 10번, 4번 인덱스에 저장이 되겠다.

0	1	2	3	4	5	6	7	8	9	10
				15						10
				OK						OK

그렇지만 21를 삽입하려면 10번 인덱스에 이미 값이 저장되어 있기 때문에 다음 인덱스인 0번을 살펴봐야겠다.(C는 Collapse(충돌)의 약자로 생각하길 바라면서...)

0	1	2	3	4	5	6	7	8	9	10
21				15						10
OK										C

그리고 26을 삽입하게 된다면 분명 4번 인덱스에 15가 떡하니 앉아있으니깐 다음 인덱스인 5번에 삽입을 하도록 작동한다.

0	1	2	3	4	5	6	7	8	9	10
21				15	26					10
				C	OK					

물론 31를 삽입하게 된다면 정상적으로 9번 인덱스에 삽입이 된다. 하지만 37를 삽입하게 된다면 어떤 일이 벌어질까...?? 4번 인덱스에 15가 떡하니 앉아 있으니 다음 인덱스인 5번 인덱스를 살펴보는데 26이 떡하니 있으니깐 6번 인덱스에 자리를 채워주는 방법이 최선이다.

0	1	2	3	4	5	6	7	8	9	10
21				15	26	37			31	10
				C	C	OK			OK	

그렇지만 이러한 방법을 잘만 이용하면 효율적이지만, 15랑 26이랑 31이 모여 있는 자리를 살펴보면 같은 나머지들끼리 유유상종 모여 있는 모습을 살펴볼 수 있다. 이것이 바로 1차 군집 문제라고 한다. 충돌이 발생한 근처에 데이터가 쌓이니깐 이러한 문제를 해결하는 방법은 다음을 살펴보도록 하고 일단 나머지를 이용한 간이적으로 충돌을 막을 수 있는 방법에 대해서 코딩을 참고해보자.

```
public class Hash_03{
  static class HashTable{
    static final int SIZE=11;
    static final int EMPTY=0;
    static final int DELETED=-1;
    int[] a;
  public HashTable(){ a=new int[SIZE]; }
  public void add(int value) throws Exception{
    int baseIndex=value%SIZE; // 1
    int cCount=0; // 2
    // 다음 페이지 참고
```

```
// add 메소드에서 이어집니다.
    do{
      int index=(baseIndex+cCount)%SIZE;
      if(a[index]==EMPTY || a[index]==DELETED){
        a[index]=value;
        return;
      } else if(a[index]==value) return;
      ++cCount;
    }while(cCount<a.length); // 3
    throw new Exception("공간 부족"); // 4
  }
  public void remove(int value){
    int baseIndex=value%SIZE; // 1
    int cCount=0; // 2
    do{
      int index=(baseIndex+cCount)%SIZE;
      if(a[index]==EMPTY) return;
      else if(a[index]==value){
        a[index]=DELETED;
        return;
      }
      cCount++;
    }while(cCount<a.length); // 5
  }
  public boolean contains(int value){
    int baseIndex=value%SIZE; // 1
    int cCount=0; // 2
    do{
      int index=(baseIndex+cCount)%SIZE;
      if(a[index]==EMPTY) return false;
      else if(a[index]==value) return true;
      ++count;
    }while(cCount<a.length); // 6
  }
 }
 public static void main(String[] args){
    HashTable hash=new HashTable();
    int maxValue=50;
    int[] data={10, 15, 21, 26, 31, 37};
    for(int k=0;k<data.length;k++)
      hash.add(data[k]);
    for(int k=0;k<=maxValue;k++)
      if(hash.contains(k)) System.out.println(k);
}
```

1, 2 : 우선 데이터가 같은 인덱스에 충돌하였을 경우에는 cCount(약자로는 collapse count, 충돌 횟수)를 고려해줘야 하는 점이 관건이다. 그리고 본래 저장해야 하는 인덱스를 baseIndex를 통해서 나머지 연산자로 구해주는데 아까와 같이 같은 인덱스에 충돌이 일어나는 경우에 3번 문장을 살펴봐야겠다.

3 : 여기서 EMPTY랑 DELETED는 각각 빈 공간(0), 삭제된 공간(-1)를 상수로 표현해뒀다. 무엇보다 중요한 것은 do~while문을 통해서 빈 인덱스를 찾아내면서 자리가 있다면 cCount를 1씩 올려줘서 자리를 채우도록 하는 역할을 해준다. 참고로 4번 문장을 써준 이유는 해시 테이블의 자리가 꽉 차게 된다면 cCount가 배열의 크기를 넘으면 안 되니깐 넘게 되면 예외 처리를 해준다.

5 : 삭제를 하는 과정은 아까 add 메소드랑 별반 다르지는 않은데 값이 저장된 경우를 따져서 값이 있다면 DELETED를 통해서 삭제를 처리해주는 역할로 생각하면 되겠다.

6 : contains를 아까처럼 써먹으면 충돌된 데이터에 대해 고려를 하지 못하기 때문에 baseIndex랑 cCount를 통해서 리믹스를 해줘야겠다. 그렇다고 모든 테이블을 순차 탐색을 해서 시간 복잡도를 O(n)으로 바꾸면 해시 테이블의 원칙이 아니기 때문이다.

2. 저장할 위치 계산하기

충돌을 하는 경우에 저장하는 위치를 계산하는 방법은 크게 2가지로 나뉜다. 아까 우리가 살펴본 방법이 바로 Division Method(한글로 직역하면 나머지 공법)이 있고, 또 하나는 임의의 상수를 곱해줘서 삽입하는 Multiplication Method(곱하기 공법)으로 나뉘어 볼 수 있겠다.

2-1) Division Method

저장하는 배열의 크기로 나눈 나머지가 값을 저장하는 위치를 정해주는 역할을 한다. 해시 테이블을 구현한다면 대부분 이 방안을 이용한다. 배열의 값이 소수(Prime Number)인 경우에 골고루 분포를 할 수 있다. 이를 이해하는 방법은 이따가 충돌을 해결하는 방안 중 Double Hashing에서 살펴보도록 하겠다. 일단 그렇다는 점만 알아두도록 하자.

2-2) Multiplication Method

우리가 해외여행을 간다면 달러, 엔화, 위안 등등으로 환전을 하고 가야 기념품을 사오든 밥을 먹든 할 수 있다. 이처럼 상수 A라는 녀석을 0부터 1까지 임의의 소수로 지정을 하고 저장을 할 값과 곱한 이후에 소수만 얻어 와서 다시 배열의 크기를 곱해서 자리를 결정하는 원리로 생각하면 되겠다. 가령 예를 들어 A를 0.1111로 잡고(소수 곱셈이 워낙 복잡하니깐^^) 배열의 크기를 16으로 잡아서 데이터 20을 저장하는 자리를 계산하는 과정을 살펴보자.

-> 일단 20이랑 0.1111를 곱하면 2.222가 나오게 된다.

-> 그럼 여기서 정수부인 2는 필요 없고 궁극적으로 필요로 한 0.222를 가져와서 배열의 크기인 16을 곱하게 된다면 3.552가 나오게 되면서 소수점을 버린 3이 20이 저장되는 인덱스로 볼 수 있게 된다.

배열의 크기는 나머지로 따지는 원리가 아니기 때문에 무엇이든 상관이 없는데 대부분 2의 배수로 결정을 하는데 CPU 캐시 hit ratio랑 메모리 관리 측면에서(컴구 시간에 배웠을 것이다.) 2의 배수가 그냥 편하다. 근데 나머지 공법보단 계산 과정이 복잡하다...

3. 충돌을 해결하는 방법

충돌을 피하는 방법에 대해서 살펴본다면 아까 이용한 방법이 바로 Linear Probing(한글로 직역하면 선형 해결법??)인데 이를 포함한 충돌을 해결하는 방안은 크게 4가지로 나뉜다.

-> Chaining : 저장할 데이터에 대해서 연결 리스트(Linked List)로 등록하는 방법.

-> Open Addressing : 그 다음 칸에 저장을 하는 방법인데 이는 아까 설명한 Linear Probing, Quadratic Probing, Double Hasing 3가지로 나뉘게 된다.

3-1) Chaining 공법 -> 다음 페이지로

우리가 예를 들어 몸이 안 좋아서 감기약이나 두통약을 먹는다고 가정을 하자. 요즘 무슨소 가게에서 요일별로 약을 나눠 먹을 수 있는 케이스가 있는데 감기뿐 만이 아니라 혈압약, 항생제 등등을 먹어야 하는 경우도 있어서 같은 요일 같은 시간에 보관을 하는 원리로 생각을 하면 쉽다.(비록 약을 혼합해서 먹으면 무시무시한 일이 일어나니 알아서 유두리있게...)
Chaining 공법은 강의 노트에 있는 코드를 참고해보면 아래와 같이 작성이 가능하다.

```
public class Hash_04 {
    static class Node {
        int value;
        Node next;
        public Node(int value, Node next) {
            this.value = value;
            this.next = next;
        } // 1
        static Node remove(Node node, int value) {
            if (node == null) return null;
            if (node.value == value) return node.next;
            node.next = remove(node.next, value);
            return node;
        } // 2
        static boolean contains(Node node, int value) {
            if (node == null) return false;
            if (node.value == value) return true;
            return contains(node.next, value);
        } // 3
    }
    static class HashTable {
        static final int SIZE = 11;
        Node[] a;
        public HashTable() {
            a = new Node[SIZE];
        }
        public void add(int value) throws Exception {
            int index = value % SIZE;
            a[index] = new Node(value, a[index]);
        } // 1
        public void remove(int value) {
            int index = value % SIZE;
            a[index] = Node.remove(a[index], value);
        } // 2
        public boolean contains(int value) {
            int index = value % SIZE;
            return Node.contains(a[index], value);
        } // 3
    }
    public static void main(String[] args) throws
Exception {
        int maxValue = 200, maxCount = 10;
        HashTable hashTable = new HashTable();
        int[] data = { 10, 15, 21, 26, 31, 37 };
        for (int i = 0; i < data.length; ++i)
            hashTable.add(data[i]);
        for (int i = 1;  i <= maxValue; ++i)
            if (hashTable.contains(i))
                System.out.println(i);
    }
}
```

1 : 여기서 Node에 add 메소드를 굳이 안 쓰는 이유는 무엇인가? 이는 값이 들어오면 앞으로 저장을 하는 원리로 살펴볼 수 있겠다. Chaining을 이용하면 충돌 횟수에 대해서 고려를 안 하게 되니깐 편리하다는 점을 느낄 수 있다.
2, 3 : 또한 삭제와 포함도 마찬가지로 이처럼 작성을 하면 되는데 아까 차이를 본다면 충돌 횟수를 안 따지는 정도??

Chaining 메소드를 이용해서 작성을 하게 된다면 아래와 같은 결과가 나오게 된다.

0	
1-3	... 중략 ...(데이터 X)
4	37 / 26 / 15
5-8	... 중략 ...(데이터 X)
9	31
10	21 / 10

3-2) Linear Probing
아까 해시 테이블에서 충돌을 해결하는 방안에서 이용했던 공법이기 때문에 여기서는 코드를 작성하지 않고 넘어가자. 이는 본래 저장할 위치와 충돌 횟수를 통해서 결정이 되는 것은 사실인데 충돌 횟수를 1씩 더해주는 원리로 작용한다. 그런데 만일 나머지가 같은 데이터들이 충돌해서 유유상종으로 모여 있는 경우에는 1차 군집 문제가 발생하게 된다. 1차 군집은 선형 문제 해결에서 나오는 문제점으로 살펴보면 되겠다. 쉽게 이야기해서 아침에 고속도로를 타는데 양재IC에서 차 막히는 경우를 생각해볼 수 있을까??

3-3) Quadratic Probing
Quadratic는 영어로 2차 방정식이란 뜻이다. 이는 아까 Linear Probing에서 충돌 횟수를 1씩 더해줬었다면, 여기서는 1, 4(2^2), 9(3^2), 16(4^2) ... 씩 뒤로 밀어내는 원리로 생각하면 된다. 아까 1차 군집 문제를 해결하기 위한 우선적인 대처법으로 생각을 할 수 있다. 아까 2~3페이지에 있는 add 메소드를 하나 골라내서 사례를 소개하면 아래와 같이 쓸 수 있다.

```
public void add(int value) throws Exception{
    int baseIndex=value%SIZE;
    int cCount=0;
    do{
  int index=(baseIndex+cCount*cCount)%SIZE;// *
      if(a[index]==EMPTY || a[index]==DELETED){
        a[index]=value;
        return;
      } else if(a[index]==value) return;
      ++cCount;
    }while(cCount<a.length);
    throw new Exception("공간 부족");
}
```

여기서 밑줄 친 부분이 제곱을 더해주는 역할로 생각을 할 수 있다. 아까 삽입한 데이터들을 사례로 어떻게 돌아가는지에 대해 간단하게 짚고 넘어가자.

우선 10과 15는 그대로 저장된다.

0	1	2	3	4	5	6	7	8	9	10
				15						10
				OK						OK

그렇지만 21를 삽입하게 된다면 10에서 충돌이 일어나게 된다. 그럼 1의 제곱인 1를 더해서 다음 자리인 0번째에 저장이 된다.

0	1	2	3	4	5	6	7	8	9	10
21				15						10
OK										C

물론 26도 마찬가지로 충돌이 일어나게 되니깐 1의 제곱인 1를 더해서 다음 자리인 5번째에 저장된다.

0	1	2	3	4	5	6	7	8	9	10
21				15	26					10
				C	OK					

31은 아까처럼 9번 인덱스에 저장이 된다.

0	1	2	3	4	5	6	7	8	9	10
21				15	26				31	10
									OK	

그렇지만 37를 삽입하게 된다면... 4번 인덱스에 저장을 해야 되는데 이미 15가 떡하니 앉아있다. 1의 제곱인 1만큼 다음 인덱스인 5에 저장을 해야 되는데 26가 떡하니 앉아 있다. 충돌 횟수가 2번이 되어서 2의 제곱인 4만큼 이동한 8번 인덱스로 이동을 해서 저장을 함으로서 마무리한다.

0	1	2	3	4	5	6	7	8	9	10
21				15	26			37	31	10
				C	C			OK		st

물론 나머지가 같은 경우에도 Linear Probing보다 몰리는 경우가 줄어들게 되는 것을 여기서는 사례로서 별 차이가 없지만 실질적으로 방대한 테이블로 보면 연못처럼 띄엄띄엄 멀어져서 군집의 크기가 줄어드는 현상을 볼 수 있다. 그렇지만 1차 군집에서 충돌이 발생하면 모두 동일하게 4칸을 건너게 됨으로서 2차 군집 문제에 대해서는 보장을 하지 못한다는 단점이 있다. 이를 해결하기 위해서 건너 뛰는 폭이 매번 다르게 조정을 할 수 있는 Double Hashing을 소개한다.

3-4) Double Hashing

해시 테이블은 충돌이 발생하였을 경우에 건너뛰는 폭을 매번 다르게 해서 띄엄띄엄 저장을 하는 것이 궁극적인 목표이다. 그래서 저장하는 값에 따라서 건너뛰는 폭을 다르게 계산하는 Double Hashing을 소개한다.

1단계) 처음에 넣어야 하는 인덱스를 계산해준다.
이는 마찬가지로 (삽입할 값)%(배열 크기)를 해주면 되는데 배열 크기는 소수(Prime Number)일수록 골고루 분포할 수 있다.

2단계) 충돌이 발생하는 경우에 건너뛰어야 하는 폭을 정해주는 방법을 고려해야겠다. 예를 들어 step 이란 변수를 이용해서 결정을 해주는 방법이 있겠다.
int step = (삽입할 값)%(임의의 소수) + 1;
여기서 임의의 소수는 배열의 크기보다 작은 것이 좋은 것을 염두하고 넘어가면 좋겠다. 예를 들어 테이블의 크기가 11인 경우에는 2, 3, 5 중에서 쓰는 것을 사례로 들 수 있겠다.

예를 들어 아까 10, 15, 21, 26, 31, 37를 크기가 11인 해시 테이블에 넣는 과정을 이용해서 step의 임의 소수를 5로 잡고 어떻게 돌아가는지에 대해 작성을 해보겠다.

우선 10이랑 15는 그대로 저장이 되는 건 두 말해도 잔소리일 것이다.

0	1	2	3	4	5	6	7	8	9	10
				15						10
				OK						OK

그럼 그 다음에 21를 삽입해보도록 하자. 21은 나머지가 10이라서 10번 인덱스에 넣는 방안이 있지만 이미 10이 떡하니 앉아있기 때문에 step을 이용해서 조정을 해주면 21%5+1를 하면 2가 된다. (10+1*2)%11를 해주면 1번째 인덱스로 넣으면 된다.

0	1	2	3	4	5	6	7	8	9	10
	21			15						10
	OK									C

그 다음에 26을 삽입해보면 본래 4번째 인덱스에 저장을 해야 되는데 이미 15가 떡하니 앉아 있기 때문에 step을 계산해서 step을 계산하면 26%5+1를 하니 2가 나온다. 그럼 (4+1*2)%11를 하게 되면 6번째 인덱스에 넣어주면 되겠다.

0	1	2	3	4	5	6	7	8	9	10
	21			15		26				10
				C		OK				

물론 31를 삽입하는데 있어서 9번째 인덱스에 넣으면 되니깐 큰 문제는 없다.

0	1	2	3	4	5	6	7	8	9	10
	21			15		26			31	10
									OK	

그 다음 37를 삽입하게 되면 4번째 인덱스에 원래 저장을 하면 되는데 15가 떡하니 앉아 있기 때문에 step을 계산하면 37%5+1를 하면 3이 나온다. 그럼 (4+1*3)%11를 하게 되면 7번째 인덱스에 앉히면 되겠다.

0	1	2	3	4	5	6	7	8	9	10
	21			15		26	37		31	10
				C			OK			

이처럼 Double Hashing 공법을 이용하면 충돌로 인한 데이터 군집 문제를 적게 발생하는 모습을 볼 수 있다. 해시 테이블을 이용하게 된다면 이를 이용해서 군집 문제 따윈 개한테 줘버리면 되겠다. 이번에는 add 메소드를 사례로 Double Hashing 공법이 어떤 원리로 돌아가는지에 대해 알아보자.

```
public void add(int value) throws Exception{
    int baseIndex=value%SIZE; // SIZE는 11.
    int step=(value%STEP_SIZE)+1;
    // STEP_SIZE는 5로 잡는다.
    int cCount=0;
    do{
  int index=(baseIndex+step*cCount)%SIZE;// *
      if(a[index]==EMPTY || a[index]==DELETED){
        a[index]=value;
        return;
      } else if(a[index]==value) return;
      ++cCount;
    }while(cCount<a.length);
    throw new Exception("공간 부족");
}
```

밑줄 친 부분을 참고해보면 충돌 횟수만큼 step을 곱해줘서 데이터들이 띄엄띄엄 존재하도록 하는 역할을 해준다고 생각하면 되겠다. 이제 여기서 해시 테이블의 크기가 소수여야 하는 이유에 대해서 알고 넘어가면 좋겠다. 잠깐 수학 드립이 나오겠지만 이해를 시켜주기 위함으로 생각하고 양해를 구한다.
-> Next Page ->

4. 해시 테이블과 소수의 연결고리

해시 테이블의 크기는 방금 전처럼 11이라고 잡고 시작 인덱스를 1로 가정을 하고 충돌 한 번에 6칸씩 이동을 한다고 가정을 하자. 코딩은 아래와 같이 할 수 있겠다.

```
public static void main(String[] args){
    int hashTableSize=11; // 여기
    int step=6;
    int startIndex=1;
    int[] a=new int[hashTableSize];
    for(int k=0;k<hashTableSize;k++){
    int index=(startIndex+(step*k))%hashTableSize;
      a[k]=index;
    }
    System.out.println(Arrays.toString(a));
}
```

배열의 크기랑 step의 공약수가 1이기 때문에 이에 대해서는 서로소라고 칭한다. 이처럼 해시 테이블의 크기랑 step이 서로소인 경우에는 문제없이 모든 인덱스를 방문을 할 수 있다.

하지만 여기라고 써져 있는 크기를 12로 바꾸면 어떻게 될까? 방금 문장은 0부터 10까지 골고루 나오게 되는데 바꾸게 되면 1이랑 7밖에 안 나온다. 여기서 12랑 6의 최대 공약수는 6이 나오게 되어 서로소가 아니기 때문에 1 위치에서 시작을 해서 계속 6씩 건너 뛰어 1, 7 이렇게 방문하는 수밖에 없다.

이를 통해서 살펴보면 무엇보다 중요한 것은 건너뛰는 step, 해시 테이블 크기는 서로소이어야 효율적으로 돌아가는 것을 느낄 수 있을 것이다. 어차피 step은 Division 공법이든 Multiplication 공법이든 나머지는 0부터 부지기수하게 나오지만 해시 테이블의 크기가 소수이면 서로소로 나오기 때문에 크게 상관없다. 여기서 서로소는 두 수의 공약수가 1인 경우에 해당된다.

5. hashCode() 메소드와 해시 테이블 추가 기능 소개

우리가 여태껏 공부한 내용은 int 형으로만 의존했기 때문에 큰 의미가 없다. 하지만 String이나 Wrapper 클래스, Object 클래스 등등 부지기수한 객체를 해시 테이블에 저장하기 위해서 어떤 의미를 줘야 할까? 바로 Object 클래스에 있는 hashCode() 메소드와 equals() 메소드가 있다. 비록 시험에는 안 나오는 각이여도 나중에 자바를 심화로 공부하고 싶은 사람들에게 소개하고자 한다.

5-1) hashCode() 메소드

우리가 학교를 다니면서 개개인 별 식별을 용이하게 하기 위하여 학번으로 구분을 하거나 밖에서 술이나 담배를 사러 갈 때 신분증을 확인할 때 쓰이는 주민등록번호도 해시 코드랑 비스무리한 개념으로 살펴볼 수 있겠다. 이처럼 객체를 구별하기 위하여 고유한 int 형 정수 값으로 출력시켜주는 메소드가 바로 hashCode()이다. 이는 객체의 참조 주소를 반환하는 원리로 생각하면 되겠다. 마치 주민등록번호로 치면 앞자리 6자리는 생일이고, 뒷자리 1번째는 성별을 구분하고, 2~3번째 자리는 출생 지역을 구분하는 원리로 적용하는 것과 똑같다고 생각할 수 있겠다. 여기서 객체 별로 쓰이는 hashCode()는 지난번 과제인 Person 클래스를 통해서 잠깐 되짚고 넘어가자.

```
public class Person {
    String name;
    int age;
    public Person(String name, int age) {
            super();
            this.name = name;
            this.age = age;
    }
    // getter, setter, constructor는 생략하겠음.
    @Override
    public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + age;
      result = prime * result + ((name == null) ? 0 :
name.hashCode());
      return result;
    } // 1
    @Override
    public boolean equals(Object obj) {
      if ((obj instanceof Person) == false)
        return false;
      Person p = (Person)obj; return (this.name == null ?
p.name == null : this.name.equals(p.name)) && this.age ==
p.age;
    } // 2
}
```

여기서 살펴봐야 하는 메소드는 1번이 되시겠다. 이 코드는 Eclipse를 가지고 있으신 분들이라면 Source 메뉴에 Generate hashCode() and equals()를 눌러주면 손쉽게 만들어진다. 하지만 지난번 강의 노트를 참고하면 Objects.hash(name, age)만으로도 쉽게 해결할 수 있다. 그렇지만 이러한 기능을 알아두면 도움이 될 거 같아서 소개해 올렸다. 여기서 1번 메소드를 보면 아까와 마찬가지로 소수(Prime Number)를 통해서 인간의 이름과 나이를 통해 해시 테이블에 저장하는데 미치는 해시 코드를 얻어옴으로서 효율적으로 저장하도록 도와주는데 이를 적용하기 위해서는 equals() 메소드를 꼭 재정의 해야 한다는 점을 명심하도록 하자. 이를 안 하게 된다면 해시 테이블에 같은 데이터가 들어 있음에도 불구하고 또 다른 곳에 저장하면 해시 테이블로서의 의미가 없기 때문이다.

5-2) Multiplication 공법에서 해시 테이블의 길이 증가 시키는 메소드

Division 공법에서 해시 테이블을 증가시키는 방법에 대해서는 소수가 부지기수하게 많기 때문에 자바의 힘으로는 역부족이다. 하지만 Multiplication 공법에서 쓰이는 해시 테이블의 크기가 짝수이면 되는 조건이 있으니 이에 대해서는 부담 없이 적용이 가능하다. 아래의 메소드를 잠깐 살펴보고 넘어가자.

```
private void resize(){
   int newSize=a.length*2;
   HashTable hash=new HashTable(newSize);
   for(int k=0;k<a.length;k++)
     if(a[k]!=EMPTY&&a[k]!=DELETED)
       hash.add(a[k]);
   this.a=hash.a;
   this.threshold=hash.threshold;
}
```

이 메소드를 이용하면서 해시 테이블에 들어있는 원소의 수들이 threshold(해시 테이블에 들어있기 좋은 원소 수의 기준 값. 강의 노트를 참고해보면 해시 테이블의 원소 수의 70%를 유지하는 것이 좋겠다.)보다 많아지게 된다면 저절로 2배로 늘려주는 메소드이다. 또한 getStepDistance 메소드를 통해서 원소를 추가 할 때 step의 크기를 소수로 조절해줌으로서 Multiplication 공법의 문제점을 보완해주는 역할을 한다.